Organization of Programming Languages
CSC 1800

Frank Klassner
Fall 2016

Why Study Programming Language **Organization**?

- Think of some programming languages you've used
- What makes a "good" programming language?
- What does "Organization" refer to? Why might this be useful to study?

Some Programming Languages

- Basic
- Pascal
- VAX11-780 Assembly
- Modula-2
- COBOL
- Ada
- SNOBOL
- Common LISP
- Prolog
- C
- C++
- Java
- Javascript
- QuickTime
- KRpano
- Objective-C
- Python

Why Study Programming Language **Organization**?

- Think of some programming languages you've used
- What makes a "good" programming language?
- What does "Organization" refer to? Why might this be useful to study?

Programming Language Popularity Measure

- TIOBE Company Rankings (www.tiobe.com)

Aug 2016	Aug 2015	Change	Programming Language	Ratings	Change
1	1		Java	19.010%	-0.26%
2	2		C	11.303%	-3.43%
3	3		C++	5.800%	-1.94%
4	4		C#	4.907%	+0.07%
5	5		Python	4.404%	+0.34%
6	7	^	PHP	3.173%	+0.44%
7	9	^	JavaScript	2.705%	+0.54%
8	8		Visual Basic .NET	2.518%	-0.19%
9	10	^	Perl	2.511%	+0.39%
10	12	^	Assembly language	2.364%	+0.60%
11	14	^	Delphi/Object Pascal	2.278%	+0.87%
12	13	^	Ruby	2.278%	+0.86%
13	11	v	Visual Basic	2.046%	+0.26%
14	17	^	Swift	1.983%	+0.80%
15	6	vv	Objective-C	1.884%	-1.31%
16	37	^^	Groovy	1.637%	+1.27%
17	20	^	R	1.605%	+0.60%
18	15	v	MATLAB	1.538%	+0.31%
19	19		PL/SQL	1.349%	+0.21%
20	95	^^	Go	1.270%	+1.19%

Why Study Programming Language **Organization**?

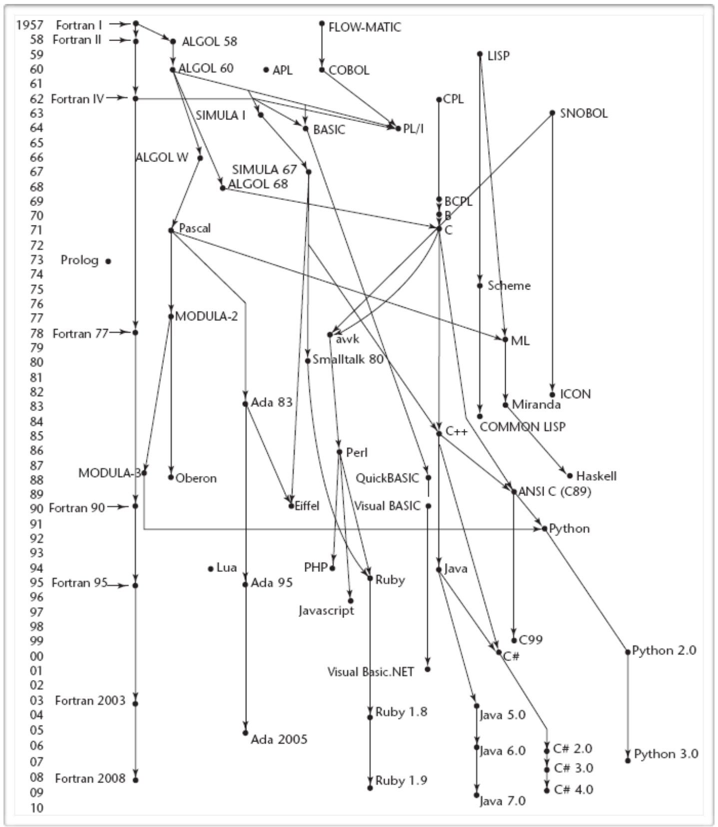
- Think of some programming languages you've used

- What makes a "good" programming language?

- What does "Organization" refer to? Why might this be useful to study?

Programming Language Design Pedigrees

See Chapter 2



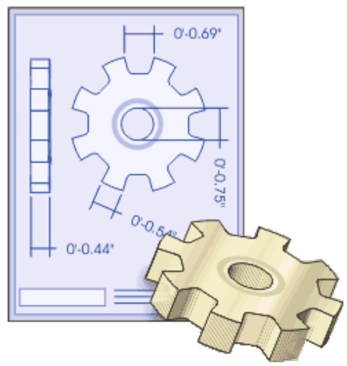
See Chapter 2

Evaluating Programming Languages

- Readability - ease with which programs can be read and understood in the context of the problem domain.
- Writability - how easily a program can created for a chosen problem domain.
- Reliability - how well a language performs to its specifications under all conditions.
- Cost - the sum of costs of training programmers, effort in writing programs, compiling programs, executing programs, relying on programs, and maintaining programs.

Qualitative PL Characteristics

- Simplicity
- Orthogonality
- Data Types
- Syntax Design
- Abstraction Support
- Expressivity
- Type Checking
- Exception Handling
- Restricted Aliasing



PL Characteristics Discussion

- Simplicity: an assessment of the number of basic constructs, any feature multiplicity, and support for operator overloading in a language.

```
// C code
a = a + 1;
a += 1;
a++
++a
```

```
;; LISP code
(setf a (+ a 1))
(incf a 1)
(1+ a)
```

PL Characteristics Discussion

- Orthogonality:
  - small set of primitive constructs can be combined in small number of ways to build control structures and data structures of language
  - every possible combination of primitives should be legal and meaningful.
  - meaning of a language feature is independent of the context of its appearance in a program

Can all data operators be applied to all data types?

In a language with hash tables and arrays (and data types int, float, char), can we put hashes into arrays and vice versa?

In C, structs can be returned by functions, but arrays cannot.

PL Characteristics

- Data Types: the adequacy of facilities for defining data types and data structures.

Can I use booleans or enumerated data types? Or am I stuck with using "special values" to represent meanings?

NULL and 0 (or empty list and NIL) are interchangeable in some languages

PL Characteristics

- Syntax Design: how complex are the forms of the language, and how obvious are the meanings of these forms?

```
Indentation in Python
can change meaning of
code
```

```
// Java Code
while (i<1000) {
  if (a[i] > 100) {
    i = i+2;
} else {
  i = i + 1;
}
 i = i + 3;
}
```

```
Ada version uses
“end if;“
to terminate if
clauses
and
“end while;”
to end while loops
```

```
perl code:

 a = a + 1 unless a < 0;

//also

if(a >= 0) a = a + 1;
```


PL Characteristics

- Abstraction Support: ability to define and use complicated structures or operations in ways that allow details to be ignored.

```
Procedures in Pascal
```

```
Methods in Java
```

```
trees and linked lists:

“natural” vs. array
implementation
```

PL Characteristics

- Expressivity: how convenient or cumbersome is it to specify computation in the language?

Boolean Short Circuit

multiple ways of expressing control decisions?

automatic declaration of accessor methods

PL Characteristics

- Type Checking: what kind of testing for type errors does the language support at compile time and runtime?
  - Java: NULL at compile or runtime is BAD!!!
  - C: NULL at compile MIGHT be bad
    - at runtime, eh, could just be treated as 0.

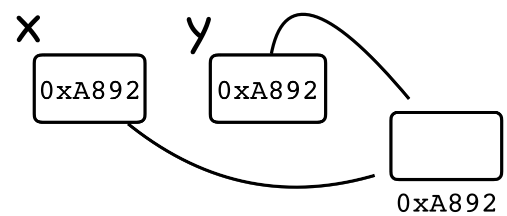
PL Characteristics

- Exception Handling: how does a language support interception of runtime errors, take corrective measures, and continue?

- Aliasing: how many restrictions does the language place on possibilities for more than one reference to the same memory cell to exist?

```
//C language:
int *x = malloc(sizeof(int));
int *y = x;

//also in C language:
int a;
int *b = &a;
```



Evaluation/Characteristic Relationships

CHARACTERISTIC	**EVALUATION CRITERIA**			
	READABILITY	WRITABILITY	RELIABILITY	COST
Simplicity	●	●	●	●
Orthogonality	●	●	●	●
Data Types	●	●	●	●
Syntax Design	●	●	●	●
Support for Abstraction		●	●	●
Expressivity		●	●	●
Type Checking			●	●
Exception Handling			●	●
Restricted Aliasing			●	●

Syntax & Semantics

Definitions

- Syntax: rules governing the form of a language's expressions, statements, and program units.
- Semantics: rules that determine the meaning of a language's expressions, statements, and program units.

- Example Showing the Relationship: Boolean Logic

Syntax

A —> B ^ ~C OK
A ~ C B —> NOT OK

Semantics

AND	A	B
f	f	f
f	f	t
f	t	f
t	t	t

Programming Language Syntax

- Sentence: string in a language
- Lexeme: simplest syntactic unit
- Token: category (type) of lexeme

if (x == 0) { y = y + 2 };

Lexeme	Token
if	branch_cmd
(	expr_lt_bound
x	identifier
==	equality_test
0	int_literal
)	expr_rt_bound

Programming Language Syntax

- Backus-Naur Form (BNF): metalanguage for programming language that uses abstractions for syntax structure.
  - Similar to Chomsky's Context-Free Grammars
  - BNF rules are called "productions"
  - Abstractions = nonterminal symbols (angle brackets)
  - Lexemes/Tokens = terminal symbols
  - BNF rules specify how symbols can be transformed

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one right hand side (RHS)
  - Use "|" symbol to combine multiple RHSs into one rule
- Can't have multiple symbols in left hand side
  - This would make the grammar "context sensitive"
- Derivative View: BNF rules explain how to create legal strings
- Parse View: Show how rules created the structure in a string

Backus-Naur Form

LHS → RHS

Derivation

<assign_stmt> → <id> = <expr>

<id> → A | B | C | D | E | F | G

<expr> → <id> + <expr>

| <id> * <expr>
| (<expr>)
| <id>

D = G * (A + B)

Backus-Naur Form

Very versatile, can describe complex organizations

Recursion & Lists

<stmtlist> → <stmt>; <stmtlist>

<stmtlist> → <stmt>;

Operator Precedence

<expr> → <expr> + <term>

<expr> → <term>

<term> → <term> * <factor>

<term> → <factor>

<factor> → (<expr>) | <id>

Extended Backus-Naur

- Convenience metasymbols (not in actual PL)
- [ ] indicates optional items
- { } shows items can be repeated 0 or more times
- (| |) shows multiple choice; one must be selected

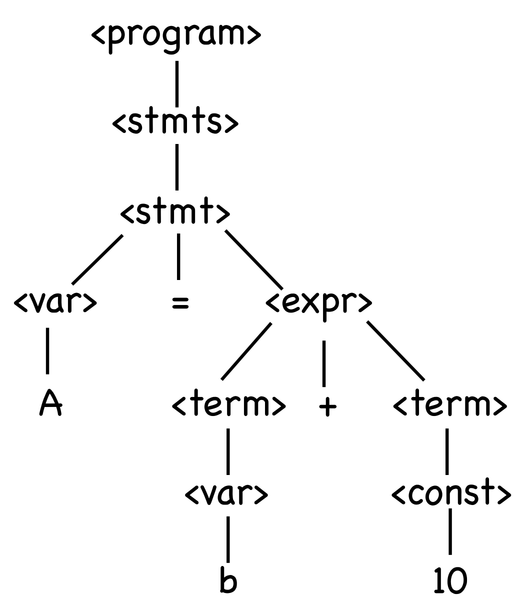
- Can we prove EBNF is no more powerful (expressive) than BNF?

Parse Trees

- Leftmost Derivations: when parsing, always expand leftmost non-terminal symbol first
- Rightmost Derivations: when parsing, always expand rightmost non-terminal symbol first

Parse Tree

- Hierarchical Representation of a derivation

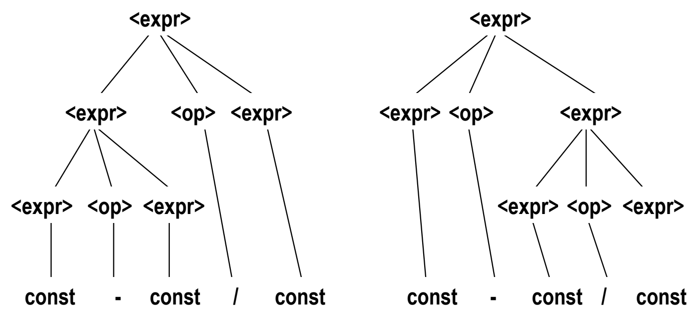


Ambiguous Parse Trees

- A BNF grammar is ambiguous if and only if (iff) it generates a sentential form (legal string) that has two or more distinct parse trees.

<expr> → <expr> <op> <expr> | const

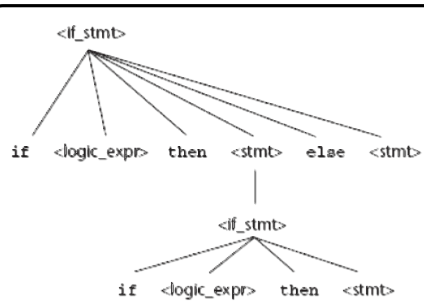
<op> → / | -

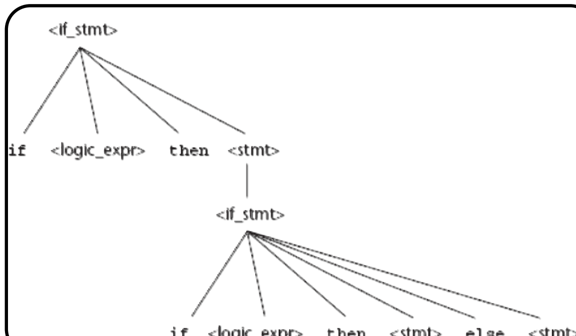


Ambiguity and IF statements

- How (not) to define IF statements with optional ELSEs.
  - <if_stmt> → if <logic_expr> then <stmt> [else <stmt>]
  - <stmt> → <if_stmt>

if <logic> then if <logic> then <stmt> else <stmt>





Non-Ambiguous IF Grammar

- <stmt> --> <matched> | <unmatched>
- <matched> --> if <logic_expr> then <matched> else <matched>
- <matched> --> <nonif_stmt>
- <unmatched> --> if <logic_expr> then <stmt>
- <unmatched> --> if <logic_expr> then <matched> else <unmatched>

Static Semantics

- Nothing to do with meaning!
- Context-free Grammars cannot describe all of the syntax of programming languages
  - Cumbersome Example: type management of operands
  - Not possible Example: variables must be declared before they are used

Attribute Grammars

- Attribute Grammars have additions to BNF/CFG notation to carry some semantic info on parse tree nodes
- Primary value:
  - Static Semantics specification
  - Compiler design (static semantics checking)

Semantics of a Program/PL

- No single widely accepted notation for describing semantics
- Why do we need any?
  - programmers need to know what statements mean
  - compiler writers must know exactly what constructs do
  - correctness proofs would be possible
  - compiler generators would be possible
  - designers could detect ambiguities and inconsistencies

Some Attempts at Formal Semantics Specs

- Operational Semantics: Describe the meaning of a program in terms of executing its statements on a machine, either simulated or actual.
  - Uses a virtual machine
  - The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement just executed.

Some Attempts at Formal Semantics Specs

- Denotational Semantics:
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the entities onto instances of the mathematical objects
  - Language constructs meanings are defined only by the values of the program's variables

Denotational Semantics

- The state of a program is the values of all of its current variables
  - $\mathbf{s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}}$
- Let VARMAP be a function that, given a variable name and a state, returns the current value of the variable
  - $\mathrm{VARMAP}(i_j, s) = v_j$

Map expressions onto $Z \cup \{\text{error}\}$

We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

Denotational Semantics

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
                OR Me(<binary_expr>.<right_expr>, s) =
                              undef)
               then error
          else
          if (<binary_expr>.<operator> == '+' then
             Me(<binary_expr>.<left_expr>, s) +
                    Me(<binary_expr>.<right_expr>, s)
          else Me(<binary_expr>.<left_expr>, s) *
              Me(<binary_expr>.<right_expr>, s)
      ...
```

Denotational Semantics

Maps state sets to state sets U {error}

$M_a(x := E, s) \; \Delta =$

if $M_e(E, s)$ == error

then error

else $s' = \{<i_1,v_1'>,<i_2,v_2'>,\ldots,<i_n,v_n'>\}$,

where for $j = 1, 2, \ldots, n$,

if $i_j == x$

then $v_j' = M_e(E, s)$

else $v_j' = VARMAP(i_j, s)$

Evaluation of Denotational Semantics

Can be used to prove the correctness of programs

Provides a rigorous way to think about programs

Can be an aid to language design

Has been used in compiler generation systems

Because of its complexity, often of little use to practical language users

Axiomatic Semantics

Based on formal logic (predicate calculus)

Original purpose: formal program verification

Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)

The logic expressions are called assertions

Axiomatic Semantics (cont.)

An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution

An assertion following a statement is a postcondition

A weakest precondition is the least restrictive precondition that will guarantee the postcondition

Axiomatic Semantics Form

Pre-, post form: {P} statement {Q}

An example

a = b + 1 {a > 1}

One possible precondition: {b > 10}

Weakest precondition: {b > 0}

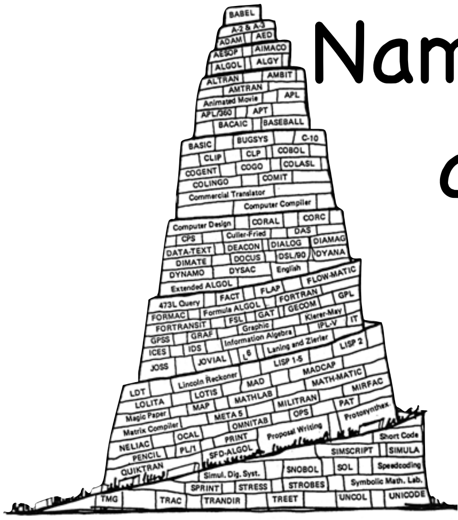
Program Proof Process

The postcondition for the entire program is the desired result

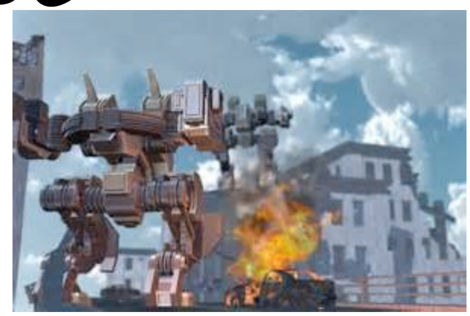
Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

```
BEGIN
  Boolean B == true;
  Procedure P;
    Print (B);
  End p;

  Begin
    Boolean B == false;
    Call P; //'STATIC' PRINTS TRUE
            //'DYNAMIC' PRINTS FALSE
  end
END
```



Names, Bindings, and Scope



It drove outside our scope of operations

Topics to Cover

- Names
- Variables
- Binding Concept
- Scope Concept
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introductory Concepts

- Imperative languages are abstractions of von Neuman architecture
  - Memory + Processor
- Variables (storage locations) have attributes:
  - name & scope,
  - lifetime,
  - type checking,
  - initialization,
  - type compatibility

John von Neuman

Names

- What strings can be used to represent a name?
  - Are names case sensitive?
  - Are there length limitations?
    - Fortran 95: max length = 31
    - C99: no limit in source code, but only first 63 are significant
    - C#, Ada, Java, Python: no limit, all are significant
    - C++, Lisp: no limit, but implementers often impose one

Names and Special Words

- Special Word: symbols that are an aid to readability
- Keyword: a word/symbol that is special only in certain contexts
- Reserved Word: a special word that cannot be used as a user-defined name
  - potential problem: if too many reserved words, then name collisions can occur

Variables

- Defn: an abstraction of a memory cell
- Characterized by a sextuple:
  - name
  - address
  - value
  - type
  - lifetime
  - scope

Variable Attributes

- Name: not all variables have them
- Address: the memory address associate with it
  - a variable may have different addresses at different times during execution
  - a variable may have different addresses at different places in a program
  - if two variable names can access the same memory location, they are called aliases
  - aliases are created via pointers, reference variables, C and C++ unions

Variable Attributes

- Type: determines the range of values and the set of operations defined for the variable.
- Value: contents of the location associated with the variable
  - l-value: address
  - r-value: value
- Abstract memory cell: physical cell or collection of cells associated with a variable

Binding Concept

- Defn: a binding is an association between an entity and an attribute, such as between a variable and its type, or value, or allowed operations, etc.
- Defn: Binding Time is the time at which a binding takes place.
  - Language design time
  - Language implementation time
  - Compile time
  - Link/Load time
  - Execution time

Static vs. Dynamic Binding

- Static: a binding is static if it first occurs before run time and remains unchanged throughout program execution
- Dynamic: if it first occurs during execution OR can change during execution of the program.

Type Binding

- How is a type specified in the programming language?
- When does binding take place?
- If static, the type may be specified by either an explicit or implicit declaration
  - explicit: stated in program source code
  - implicit: mechanism specifies through default conventions
  - BASIC, Perl, Ruby, JavaScript and PHP use implicit
  - Java, C, C++ use explicit
  - Lisp, Fortran???

Type Binding (Explicit/Implicit)

- Some languages use type inferencing to determine variable types (context)
  - C# - variable can be declared with **var** and an initial value. Initial value sets the type
- (Type/Value Binding Example) The Swift language uses
  - `let —> declares a name to have a constant value (and the value's type)`
  - `var —> declares a name to be a variable whose value can change.`

Dynamic Type Binding

- JavaScript, Python, Ruby, Lisp, PHP
- Specified through assignment statements
  - list = `(2 3 4.33 `alpha)
  - list = 17.3
- Advantage: flexibility
- Disadvantages: high cost with dynamic type checking and interpretation; type error detection by compiler is difficult

Variable Lifetimes

- Storage Bindings & Lifetime
  - Allocation: getting a cell from some pool of available cells
  - Deallocation: putting a cell back in the pool
- Lifetime of a Variable: the time during which the variable is bound to a particular memory cell

Lifetime Categories

- Static: bound to memory cells before execution begins, and remains bound to same cell throughout execution
  - Advantage: efficiency (direct addressing)
  - Disadvantage: lack of flexibility (no recursion)

Lifetimes

- Stack-Dynamic: Storage bindings are created for variables when their declaration statement is elaborated (e.g. executed)
- If scalar, all attributes except address are statically bound
  - advantage: allows recursion, conserves storage
  - disadv: overhead of allocation and deallocation; inefficient references (indirect addressing)

Lifetimes

- Explicit heap-dynamic: allocation and deallocated by explicit directives during execution (e.g. “new”)
  - Example: (C++)

```
int *intnode;           // Create a pointer variable
intnode = new int; // Allocate an int’s space on heap
.....
delete intnode;  // Deallocate the heap-dynamic variable
                       // to which intnode points
```

  - Advantage: dynamic storage management
  - Disadvantage: occasionally unreliable

Lifetimes

- Implicit heap-dynamic: allocation and deallocation caused by assignment statements
  - Examples (Lisp & Javascript, not equivalent)

```
(setf a (list 1 ‘x ‘y 10.0))
```

```
a  = [1 “x” “y” 10.0]
```

  - advantage: flexibility
  - disadvantages: inefficient, all attributes are dynamic, loss of error detection

Scope

- Scope of a variable is the range of statements over which it is visible
- local variables of a program unit are those declared in that unit
- nonlocal variables of a program unit are those that are visible in the unit but not declared there
- Global Variables are a special category of nonlocal variables
- A language's scope rules determine how references to names are associated with variables

Static Scope

- Based on program's text (source code)
- To connect a name reference to a variable, the compiler must find the declaration
  - Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Variables can be hidden from a unit by having a "closer" variable with the same name

Dynamic Scope

- Based on calling sequences of program units (methods, functions, etc), not their textual layout.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to the current point.

Scope Example

```
function big() {

    function sub1()
       var x = 7;
    function sub2() {
      var y = x;
    }
   var x = 3;
 }
```

Static scoping

Reference to x in sub2 is to big's x

Dynamic scoping

Reference to x in sub2 is to sub1's x

Lexical Analysis and Parsing